Sequential hardware, "language philosophy" and first steps in C++

Scientific Computing Winter 2016/2017

Lecture 2

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

With material from “Introduction to High-Performance Scientific Computing” by Victor Eijkhout (`http://pages.tacc.utexas.edu/~eijkhout/istc/istc.html`) and `http://www.cplusplus.com/`

$\sim$

Admin (Recap from last time)

Me

- Name: Dr. Jürgen Fuhrmann (no, not Prof.)
- Contact: `juergen.fuhrmann@wias-berlin.de`
- Lectures:
  Mon: 8-10 FH 311 or UNIX Pool
  Thu: 10-12 FH 301
- Consultation: **Mon 10-12 FH 303**
  More at WIAS on appointment
- Affiliation: Weierstrass Institute for Applied Analysis and Stochastics, Berlin (WIAS); Deputy Head of *Numerical Mathematics and Scientific Computing*
- Experience/Field of work:
  - Numerical solution of PDEs
  - Development, investigation, implementation of finite volume discretizations for nonlinear systems of PDEs
  - Ph.D. on multigrid methods
  - Applications: electrochemistry, semiconductor physics, groundwater...
  - Software development:
    - WIAS code pdelib (`http://pdelib.org`)
    - Languages: C, C++ , Lua, Fortran (still sometimes), Python (recently)
    - Visualization (OpenGL)

Admin stuff

- There will be coding assignments.
  - Unix pool
  - Linux on your own PC/laptop
  - MacOSX + Windows should work, but I can't support them
  - Virtual Machine anyone (Vagrant/Virtualbox)?
- Access to examination
  - Attend $\approx$ 80% of lectures
  - Return assignments (#2-3, but yet to be determined)
  - General activity during course
- Course material will be online: `http://www.wias-berlin.de/people/fuhrmann/teach.html`

Literature

- Numerical methods
  - Y. Saad: Iterative methods for sparse linear systems `http://www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf`
  - V. Eijkhout: Introduction to High-Performance Scientific Computing `https://www.tacc.utexas.edu/~eijkhout/Articles/EijkhoutIntroToHPC.pdf`
  - A. Ern, J.-L. Guermond: Theory and Practice of Finite Elements
  - R Eymard, T Gallouët, R Herbin: Finite volume methods. In Handbook of numerical analysis `https://www.cmi.univ-mrs.fr/~herbin/PUBLI/bookevol.pdf`
- C/C++: look for resources on the new standard C++11
  - B. Stroustrup: The C++ Programming Language, **4th Edition**
  - P. Gottschling: Discovering Modern C++: An Intensive Course for Scientists, Engineers, and Programmers (C++ In-Depth)
  - `http://www.cplusplus.com/`
  - `https://isocpp.org/get-started`
  - `http://en.cppreference.com/w/`
- Python: look for resources on Python3
  - `https://www.python.org/`
  - `https://docs.python.org/3/tutorial/`
  - H.P. Langtangen († 2016): A Primer on Scientific Programming with Python `https://hplgit.github.io/primer.html/doc/pub/half/book.pdf`

Intended aims topics of this course

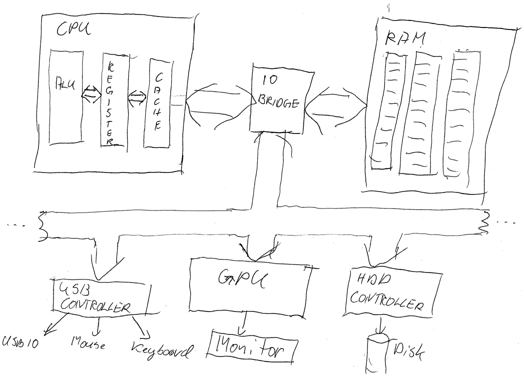
"The purpose of computing is insight, not numbers."
(`https://en.wikiquote.org/wiki/Richard_Hamming`)

- Indicate a reasonable path within the very diverse sets of tools
- Recapitulation of relevant topics from numerical analysis
- Introduction to C++ and Python and their interaction
- Provide technical skills to understand a part of the inner workings of the relevant tools
- Focus on partial differential equation (PDE) solution
  - Numerical mathematics recall
  - Finite elements
  - Finite volumes
  - Mesh generation
  - Nonlinear if time permits – so we can see some real action
  - Parallelization
  - A bit of visualization
- Tools/Languages
  - C++/Python and their interaction
  - Linux focused (but not restricted to)
  - Parallelization: Focus on OpenMP, but glances on MPI, C++ threads
  - Visualization using python tools

$\sim$

Sequential hardware

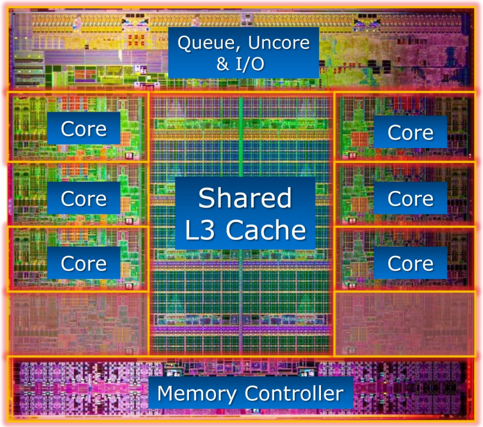
von Neumann Architecture



- Data and instructions from same memory
  - Instruction decode: determine operation and operands
  - Get operands from memory
  - Perform operation
  - Write results back
  - Continue with next instruction

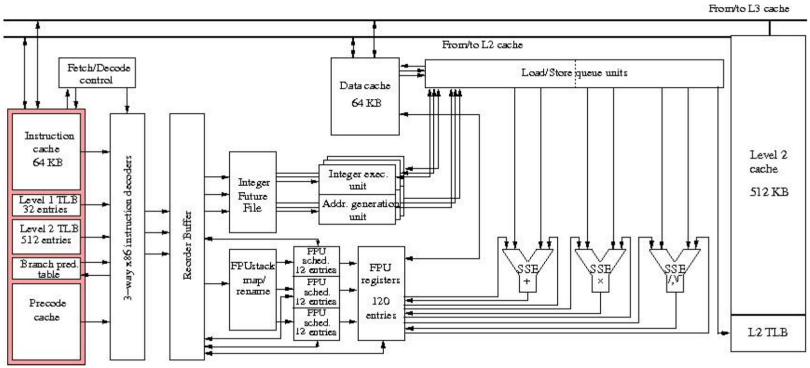
Contemporary Architecture

- Multiple operations simultaneously “in flight”
- Operands can be in memory, cache, register
- Results may need to be coordinated with other processing elements
- Operations can be performed speculatively



Modern CPU. From: https://www.hartware.de/review_1411_2.html

What is in a “core” ?



From: Eijkhout

Modern CPU functionality

- Traditionally: one instruction per clock cycle
- Modern CPUs: Multiple floating point units, for instance 1 Mul + 1 Add, or 1 FMA
  - Peak performance is several operations /clock cycle
  - Only possible to obtain with highly optimized code
- Pipelining
  - A single floating point instruction takes several clock cycles to complete:
  - Subdivide an instruction:
    - Instruction decode
    - Operand exponent align
    - Actual operation
    - Normalize
  - Pipeline: separate piece of hardware for each subdivision
  - Like assembly line

Memory Hierachy

- Main memory access is slow compared to the processor
  - 100–1000 cycles latency before data arrives
  - Data stream maybe 1/4 floating point number/cycle;
  - processor wants 2 or 3
- Faster memory is expensive
- *Cache* is a small piece of fast memory for intermediate storage of data
- Operands are moved to CPU *registers* immediately before operation
- Data is always accessed through the hierarchy
  - From egisters where possible

```
    - Then the caches (L1, L2, L3)

            - Then main memory
```

Machine code

- Detailed instructions for the actions of the CPU
- Not human readable
- Sample types of instructions:
  - Transfer data between memory location and register
  - Perform arithmetic/logic operations with data in register
  - Check if data in register fulfills some condition
  - Conditionally change the memory address from where instructions are fetched $\equiv$ “jump” to address
  - Save all register context and take instructions from different memory location until return $\equiv$ “call”

```
534c 29e5 31db 48c1 fd03 4883 ec08 e85d
feff ff48 85ed 741e 0f1f 8400 0000 0000
4c89 ea4c 89f6 4489 ff41 ff14 dc48 83c3
0148 39eb 75ea 4883 c408 5b5d 415c 415d
415e 415f c390 662e 0f1f 8400 0000 0000
f3c3 0000 4883 ec08 4883 c408 c300 0000
0100 0200 4865 6c6c 6f20 776f 726c 6400
011b 033b 3400 0000 0500 0000 20fe ffff
8000 0000 60fe ffff 5000 0000 4dff ffff
```

Assembler code

- Human readable representation of CPU instructions
- Some write it by hand . . .
  - Code close to abilities and structure of the machine
  - Handle constrained resources (embedded systems, early computers)
- Translated to machine code by *assembler*

```
    .file   "code.c"
    .section    .rodata
.LC0:
    .string "Hello world"
    .text
    ...
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    %edi, -4(%rbp)
    movq    %rsi, -16(%rbp)
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
```

Registers

Processor instructions operate on registers directly - have assembly language names names like: `eax, ebx, ecx`, etc. - sample instruction: `addl  %eax, %edx`

- Separate instructions and registers for floating-point operations

Data caches

- Between the CPU Registers and main memory
- L1 Cache: Data cache closest to registers
- L2 Cache: Secondary data cache, stores both data and instructions
- Data from L2 has to go through L1 to registers
- L2 is 10 to 100 times larger than L1
- Some systems have an L3 cache, $\sim$10x larger than L2

Cache line

- The smallest unit of data transferred between main memory and the caches (or between levels of cache)
- N sequentially-stored, multi-byte words (usually N=8 or 16).
- If you request one word on a cache line, you get the whole line
    - make sure to use the other items, you've paid for them in bandwidth
    - Sequential access good, "strided" access ok, random access bad
- Cache hit: location referenced is found in the cache
- Cache miss: location referenced is not found in cache
    - triggers access to the next higher cache or memory
- Cache thrashing
    - Two data elements can be mapped to the same cache line: loading the second "evicts" the first
    - Now what if this code is in a loop? "thrashing": really bad for performance
- Performance is limited by data transfer rate
    - High performance if data items are used multiple times

∼

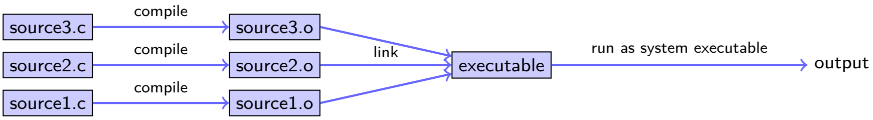
"Language Philosophy"

Compiled high level languages

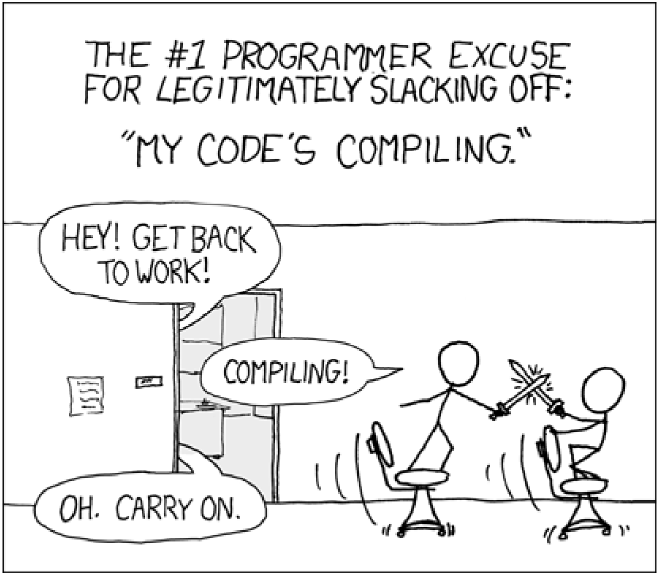
- Algorithm description using mix of mathematical formulas and statements inspired by human language
- Translated to machine code (resp. assembler) by *compiler*

```
#include <stdio.h>
int main (int argc, char *argv[])
{
  printf("Hello world");
}
```

- "Far away" from CPU $\Rightarrow$ the compiler is responsible for creation of optimized machine code
- Fortran, COBOL, C, Pascal, Ada, Modula2, C++, Go, Rust, Swift
- Strongly typed
- Tedious workflow: compile - link - run



Compiling. . .



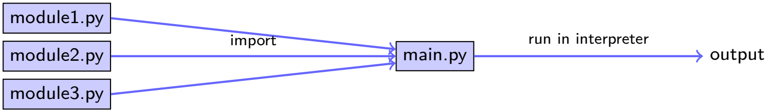
. . . from xkcd

High level scripting languages

- Algorithm description using mix of mathematical formulas and statements inspired by human language
- Need intepreter in order to be executed

```
print("Hello world")
```

- Very far away from CPU ⇒ usually significantly slower compared to compiled languages
- Matlab, Python, Lua, perl, R, Java, javascript
- Less strict type checking, often simple syntax, powerful introspection capabilities
- Immediate workflow: “just run”
  - in fact: first compiled to *bytecode* which can be interpreted more efficiently



JITting to the future ?

- As described, all modern interpreted language first compile to bytecode wich then is run in the interpreter
- Couldn't they be compiled to machine code instead? – Yes, they can: Use a just in time (JIT) compiler!
  - V8 $\rightarrow$ javascript
  - LLVM Compiler infrastructure $\rightarrow$ Python/NUMBA, **Julia** (currently at 0.5)
  - LuaJIT
  - Java
  - Smalltalk
- Drawback over compiled languages: compilation delay at every start, can be mediated by caching
- Potential advantage over compiled languages: *tracing* JIT, i.e. optimization at runtime
- Still early times, but watch closely. . .

Compiled languages in Scientific Computing

- Fortran: FORmula TRANslator (1957)
  - Fortran4: really dead
  - Fortran77: large number of legacy libs: BLAS, LAPACK, ARPACK ...
  - Fortran90, Fortran2003, Fortran 2008
    - Catch up with features of C/C++ (structures,allocation,classes,inheritance, C/C++ library calls)
    - Lost momentum among new programmers
    - Hard to integrate with C/C++
    - In many aspects very well adapted to numerical computing
    - Well designed multidimensional arrays
- C: General purpose language
  - K&R C (1978) weak type checking
  - ANSI C (1989) strong type checking
  - Had structures and allocation early on
  - Numerical methods support via libraries
  - Fortran library calls possible
- C++: *The* powerful object oriented language
  - Superset of C (in a first approximation)
  - Classes, inheritance, overloading, templates (generic programming)
  - C++11: Quantum leap: smart pointers, threads, lambdas, initializer lists in standard
  - With great power comes the possibility of great failure...

Summary

- Compiled languages important for high performance
- Fortran lost its momentum, but still important due to huge amount of legacy libs
- C++ highly expressive, ubiquitous, significant improvements in C++11

∼

First steps in C++

- most things in this part (except iostreams and array initializers) are valid C as well

Printing stuff

- IOStream library
  - “Official” C++ output library
  - Type safe
  - Easy to extend
  - Clumsy syntax for format control

```
#include <iostream>
...

std::cout << "Hello world" << std::endl;
```

- C Output library
  - C Output library
  - Supported by C++-11 standard
  - No type safety
  - Hard to extend
  - Short, relatively easy syntax for format control
  - Same format specifications as in Python

```
#include <cstdio>
...

std::printf("Hello world\n");
```

C++ : scalar data types

```
|--------------------+---------+-------+------+----------------------+----------------------|
| name               | fmt     | bytes | bits |                  min |                  max |
|--------------------+---------+-------+------+----------------------+----------------------|
| char               | %c (%d) |     1 |    8 |                 -128 |                  127 |
| unsigned char      | %c (%d) |     1 |    8 |                    0 |                  255 |
| short int          | %d      |     2 |   16 |               -32768 |                32767 |
| unsigned short int | %u      |     2 |   16 |                    0 |                65535 |
| int                | %d      |     4 |   32 |          -2147483648 |           2147483647 |
| unsigned int       | %u      |     4 |   32 |                    0 |           4294967295 |
| long int           | %ld     |     8 |   64 | -9223372036854775808 |  9223372036854775807 |
| unsigned long int  | %lu     |     8 |   64 |                    0 | 18446744073709551615 |
| float              | %e      |     4 |   32 |         1.175494e-38 |         3.402823e38  |
| double             | %e      |     8 |   64 |        2.225074e-308 |        1.797693e308  |
| long double        | %Le     |    16 |  128 |       3.362103e-4932 |       1.189731e4932  |
| bool               | %d      |     1 |    8 |                    0 |                    1 |
|--------------------+---------+-------+------+----------------------+----------------------|
```

- Type sizes are the "usual ones" on 64bit systems. The standard only guarantees that
  `sizeof(short ...) <= sizeof(...) <=sizeof(long ...)`
  - E.g. on embedded systems these may be different
- Declaration and output (example)

```
#include <cstdio>
...
int i=3;
double x=15.0;
std::printf("i=%d, x=%e\n",i,x);
```

Typed constant expressions

- C++ has the ability to declare variables as constants:

```cpp
const int i=15;
i++; // attempt to modify value of const object leads to
     // compiler error
```

Scopes, Declaration, Initialization

- **All variables are typed and must be declared**
    - Declared variables "live" in scopes defined by braces `{ }`
    - Good practice: initialize variables along with declaration
    - "auto" is a great innovation in C++11 which is useful with complicated types which arise in template programming
        - type of *lvalue* (left hand side value) is detected from type of *rvalue* (value at the right hand side)

```
{
   int i=3;
   double x=15.0;
   auto y=33.0;
}
```

Arithmetic operators

- Assignment operator

```
a=b;
c=(a=b);
```

- Arithmetic operators `+, -, *, /, %`
- Beware of precedence which (mostly) is like in math!
- If in doubt, use brackets, or look it up!

```
 +  addition
 -  subtraction
 *  multiplication
 /  division
 %  modulo
```

- Compund assignment: `+=, -=, *=, /=, %=`

```
x=x+a;
x+=a; // equivalent to =x+a
```

- Increment and decrement: `++,--`

```
y=x+1;
y=x++; // equivalent to y=x; x=x+1;
y=++x; // equivalent to x=x+1; y=x;
```

Further operators

- Relational and comparison operators `==`, `!=`, `>`, `<`, `>=`, `<=`
- Logical operators `!`, `&&`, `||`
  - short circuit evaluation:
    - if `a` in `a&&b` is false, the expression is false and `b` is never evaluated
    - if `a` in `a||b` is true, the expression is true and `b` is never evaluated
- Conditional ternary operator `?`

```
c=(a<b)?a:b; // equivalent to the following
if (a<b) c=a; else c=b;
```

- Comma operator `,`

```
c=(a,b); // evaluates to c=b
```

- Bitwise operators `&`, `|`, `^`, `~`, `<<`, `>>`
- `sizeof`: memory space (in bytes) used by the object resp. type

```
n=sizeof(char); //  evaluae
```

Addresses and pointers

- Objects are stored in memory, and in order to find them they have an *address*
- We can determine the address of an object by the & operator
  - The result of this operation can be assigned to a variable called *pointer*
  - “pointer to type x” is another type denoted by `*x`
- Given an address (pointer) object we can refer to the content using the `*` operator

```
int i=15;  // i is an object
int *j= &i; // j is a pointer
int k=*j; // now, k=15
```

- The `nullptr` object can be assigned to a pointer in order to indicate that it points to “nothing”

```
int *p=nullptr;
```

- Instead of values, addresses can be passed to functions

```
void do_multiplication(double x, double y, double *result)
{
    *result=x*y;
}
...
double x=5,y=9;
double result=0;
do_multiplication(x,y,&result) // result now contains 45
```

Pointers. . .

MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

. . . from xkcd

Functions

- Functions have to be *declared* as any other variable
- '(...) holds parameter list
    - each parameter has to be defined with its type
- type part of declaration describes type of return value
    - void for returning nothing

```
double multiply(double x, double y);
```

- Functions are *defined* by attaching a scope to the declaration
    - *Values of parameters are copied into the scope*

```
double multiply(double x, double y)
{
   return x*y;
}
```

- Functions are *called* by statements invoking the function with a particular set of parameters

```
{
    double s=3.0, t=9.0;
    double result=multiply(s,t);
    printf("s=%e, t=%e, s*t= %e\n",s,t,result); // s and t keep their values
}
```

Functions: inlining

- Function calls sometimes are expensive compared to the task performed by the function
  - The compiler may include the content of functions into the instruction stream instead of generating a call

```
inline double multiply(double x, double y)
{
   return x*y;
}
```

Flow control: Statements and simple loops

- Statements are individual expressions like declarations or instructions or sequences of statements enclosed in curly braces: `{}`:
  `{ statement1; statement2; statement3; }`
- Conditional execution: if
  `if (condition) statement;`
  `if (condition) statement; else statement;`

```
if (x>15)
    printf("error");
else
{
   x++;
}
```

- While loop:
  `while (condition) statement;`

```
i=0;
while (i<9)
{
  printf("i=%d\n",i);
  i++;
}
```

- Do-While loop: do statement while (condition);

Flow control: for loops

- This is the most important kind of loops for: numerical methods.
  for (initialization; condition; increase) statement;
  1. initialization is executed. Generally, this declares a counter variable, and sets it to some initial value. This is executed a single time, at the beginning of the loop.
  2. condition is checked. If it is true, the loop continues; otherwise, the loop ends, and statement is skipped, going directly to step 5.
  3. statement is executed. As usual, it can be either a single statement or a block enclosed in curly braces { }
  4. increase is executed, and the loop gets back to step 2.
  5. The loop ends: execution continues by the next statement after it.
- All elements (initialization, condition, increase, statement) can be empty

```
for (int i=0;i<9;i++)   printf("i=%d\n",i); // same effect as previous slide
for(;;);  // completely valid, runs forever
```

Flow control: break, continue

- break statement: “premature” end of loop

```
for (int i=1;i<10;i++)
{
   if (i*i>15) break;
}
```

- continue statement: jump to end of loop body

```
for (int i=1;i<10;i++)
{
   if (i==5) continue;
   else do_someting_with_i;
}
```

Flow control: switch

```
switch (expression)
{
  case constant1:
     group-of-statements-1;
     break;
  case constant2:
     group-of-statements-2;
     break;
  .
  .
  .
  default:
     default-group-of-statements
}
```

equivalent to

```
if      (expression==constant1) {group-of-statements-1;}
else if (expression==constant2) {group-of-statements-2;}
...
else                             {default-group-of-statements;}
```